Question 1: Will the following program (eventually) terminate? Assume that reading or writing a single variable is atomic.

\[
\begin{align*}
  i & ← 0 \\
  j & ← 0 \\
\end{align*}
\]

\textbf{thread while } i = 0 \\
\hspace{1cm} \textbf{DO } j ← j + 1 \text{ mod 2 } ; \text{ print } i \\
\hspace{1cm} \text{print } i \\

\textbf{thread while } i = 0 \\
\hspace{1cm} \textbf{do if } j = 0 \text{ then } i = 1
\]

\textbf{Answer: } No, this program will not eventually terminate. Consider the following schedule. Initially \(i, j = 0\). The first thread runs the while loop once. Now \(j = 1\). Then the next thread runs the while loop once, sees \(j = 1\) and hence does not change \(i\), i.e \(i\) stays 0. Now the first thread runs again, cycling the while loop twice. After that again \(j = 1\). And we return to the second thread. This is a schedule that creates an infinite run in which both threads take steps infinitely often.

Question 2: Will the following program (eventually) terminate, or is it possible that it runs forever? Assume that reading or writing a single variable is atomic.

\[
\begin{align*}
  a & ← 1 \\
  b & ← 1 \\
\end{align*}
\]

\textbf{thread while } a \neq 0 \\
\hspace{1cm} \textbf{do } b ← (b + a) \text{ mod 2} \\

\textbf{thread while } b \neq 0
\textbf{Answer:} Suppose $b = 1$.

Let the second thread take a single step (changing $a$ from odd to even or vice versa). If $a$ has become odd, let the first thread take two steps. Then after those two steps $b$ again is equal to 1. If $a$ has become even, let the first thread take one step. After that single step $b$ still equals 1.

This can be repeated forever (assuming $a$ is unbounded).

Alternative answer: Let second thread do one step; now $a$ is even. Then continue with the following steps forever: let the first thread do one step (because $a$ is even, $b$ remains 1), and then let the second thread do two steps so $a$ becomes even again.

P.S.: Note that the assignment $a \leftarrow 0$ is \textit{not} part of the loop body!

\textbf{Question 3:} Lamport’s logical clock algorithm works in the message passing model. Modify Lamport’s logical clock algorithm to assign logical clock values to all events in a shared memory system that supports atomic reads and atomic writes to shared memory. Prove that the logical clock created by your algorithm can be used to put the events in a total order $\langle A, \Rightarrow \rangle$ consistent with the partial order $\langle A, \rightarrow \rangle$.

\textbf{Answer:} Every node has a local counter $c_i$ as in Lamport’s algorithm, initially 0 and incremented after every action of node $i$. We assign $C_i(a) = c_i$, where $c_i$ is the value of the local counter just before action $a$ is executed.

Let $n(a) = i$ when $a$ is executed by node $i$. We define $C(a) = C_{n(a)}(a)$.

Every shared memory location $s$ is assigned a label $T_s$, initially 0. Whenever an atomic write action $w$ by node $i$ stores a value in $s$, $T_s$ is assigned $C_i(w)$. Whenever an atomic read action $r$ by node $j$ reads a value from $s$, $c_j \leftarrow \max(c_j, T_s + 1)$ and $C_j(b) = c_j$ right after that.

We define $\langle A, \Rightarrow \rangle$ by

$$a \Rightarrow b \iff \langle C(a), n(a) \rangle < \langle C(b), n(b) \rangle$$

To prove that the logical clock created by this algorithm puts the events in a total order $\langle A, \Rightarrow \rangle$ consistent with the partial order $\langle A, \rightarrow \rangle$, we have to show that for any two actions $C(a) < C(b)$ when $a \rightarrow b$.

If $a$ and $b$ are events on the same node $i$, this follows from the way counter $c_i$ is updated in between events.

If $a$ and $b$ are events on different nodes $i$ and $k$, then by the definition of $\rightarrow$ on shared memory systems there is a chain shared of variables $s_i, \ldots, s_j$ and writes $w(s)$ and reads $w(s)$ on them such that

- $a \rightarrow w(s_j)$ on the same node.
- $w(s_j) \rightarrow r(s_j)$ possibly occurring on different nodes.
• $r(s_i) \to w(s_{i+1})$ occurring on the same node.

• $r_j \to b$ on the same node.

By definition of our logical clock we have

• $C(a) < C(w(s_i))$

• $C(w(s_i)) < C(r(s_i))$

• $C(r(s_i)) < C(w(s_{i+1}))$

• $C(r_j) < C(b)$.

which proves that $C(a) < C(b)$ as required.